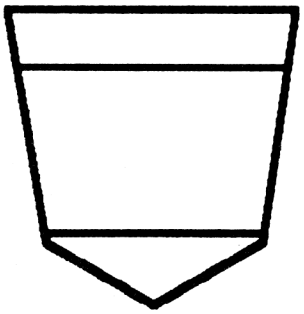
MACHINE LANGUAGE PROGRAMMING

SHARP PC-1500 & RADIO SHACK PC-2 POCKET COMPUTERS



(c) Copyright 1983 POCKET COMPUTER NEWSLETTER

MACHINE LANGUAGE PROGRAMMING

THE SHARP PC-1500

AND RADIO SHACK PC-2

POCKET COMPUTERS

It might be interesting to point out that all of the instructions discussed in the preceding section have been directives that operate on data. A programmer could organize a series of those instructions in sequential locations in memory. When told to do so, the CPU would proceed to execute each instruction. This sequential type of operation would continue for as long as there were valid instructions to be performed. The reader knows that as each instruction was dismissed with, the program counter would advance over the appropriate number of bytes to point to the next opcode. Typically, this incrementing of the address contained in the program counter is all that is needed to guide the operation of the CPU. That address value always tells the CPU where to obtain the next byte of information to be processed.

Ah, if only life could be so simple and straightforward! Take a step, advance a pointer, know where to obtain the next directive. Beautiful. But it has practical limitations. There can be no freedom of choice or ability to make decisions in such a scheme.

To empower a computer with the ability to alter its course of operation as it evaluates data, there must be an alternative to blindly following a set series of instructions. What controls the sequence in which instructions are executed? The program counter. How can this sequence be altered? By modifying the contents of the program counter!

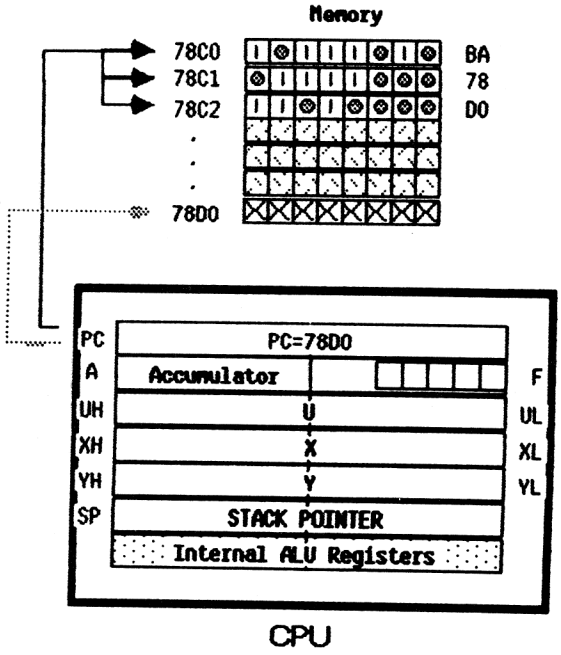
Consider what happens, for example, when a computer is started by applying electrical power. Special circuits "force" an address value into the CPU's program counter. This "tells" the CPU where to begin executing instructions in memory. If it were not for such foresight on the part of designers, the machine would theoretically start trying to execute instructions from whatever memory location randomly appeared in the program counter as power was applied. The result would likely be chaos.

Just as it is useful to be able to load the program counter with a value when a computer is started, it is also useful to be able to tell the computer to switch to another sequence of directives *whenever a programmer desires!* Again, how is this done? By simply altering the value in the program counter so that it no longer blindly points to the next sequential memory location.

A type of instruction that will perform this task is called a *jump* command. It may be used to have the computer skip over a block of instructions or to jump to a particular series of directives.

The fundamental jump directive in the LH5801 has the following mnemonic and opcode:

Mnemonic	Code
JMP nnnn	BA



CPU

Note that the instruction utilizes the addressing mode that is categorized as *direct*. That is, the opcode must be followed by two bytes of information that provide the address to which the program is to jump. Those two bytes hold the new address that is to be loaded into the program counter. *That is all this instruction does.* It does not alter the status of any of the flags. It simply causes the computer to abruptly stop executing instructions in one part of memory and start executing them in another area.

There is, however, one very important thing to remember when using the jump instruction. The address specified as the "jump to" location must contain the opcode of another instruction. That is, it must contain the first byte of code for another valid machine language command. Failure to comply with this requirement has been the woe of many an unfortunate programmer!

Artificial Intelligence

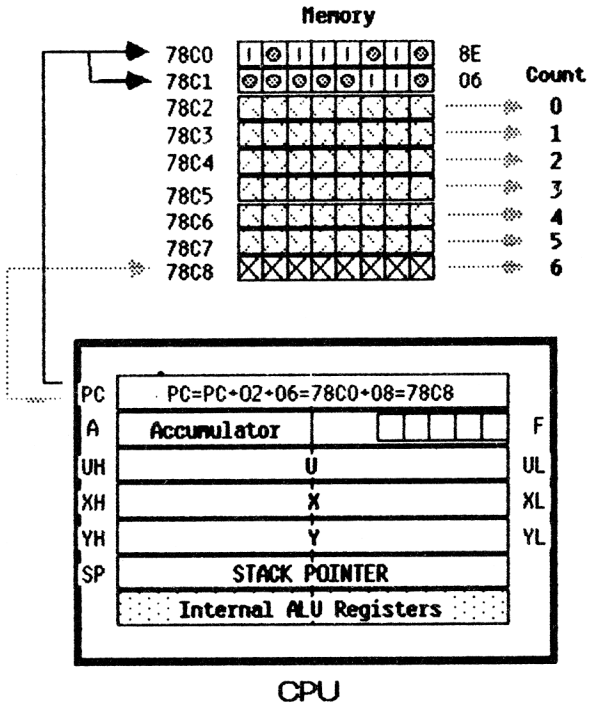
The concepts underlying the group of instructions that will be presented now are some of the most exciting in the computer world. It is this group of instructions that give a computer the attribute of being able to make decisions *and to modify its behavior as a result of those choices!* They form the basis for theoretical models that may someday lead to computers exhibiting actual artificial intelligence. Some people argue that rudimentary models of such behavior are already possible. Read on and learn how a computer mimics this power.

The LH5801 CPU has an instruction that may be considered as a variation of the jump directive. It is referred to as a *branch* command. The essential difference between it and a jump directive is that it only takes up two bytes of memory instead of three. One byte for the opcode, a second for a *relative offset* value. What is this relative offset value? It is the number of bytes, forward or backward, that the CPU is to skip before starting to execute instructions again!

Of course you can see right away that this is in effect a jump directive that is limited in scope. It is limited because the one-byte offset value means that the number of locations that can be skipped is restricted to the maximum value that can be represented in an 8-bit register. (Remember, the opcode for a true jump directive is followed by a two-byte address that can specify *any* location within the 64 kilobytes of memory that could theoretically be assigned to a LH5801 processor.) It turns out, however, that in practical applications, most jumps are relatively short -- they only need to skip over a few instructions -- *so it is worthwhile in terms of memory conservation* to use a two-byte instruction and *branch* to a new, relatively close location, rather than always having to use a three-byte jump directive. This will become more apparent as you gain machine language programming experience.

How does the branch directive actually accomplish its goal of skipping over a block of memory? You guessed it. It simply adds (or subtracts) the relative offset value to the current value in the program counter, thereby changing the program counter to point to a new location!

The key to the successful use of branch directives is to thoroughly understand how to specify the number of skipped bytes. When doing a *forward branch* you start counting with the memory location that immediately *follows* the two-byte branch directive. But, you start counting at *zero!* For instance, if the forward branch opcode (8E) was stored at address 78C0 (hexadecimal, remember) with the offset value in 78C1, and the program was to skip over 7 bytes of code, so that the next instruction executed was at location 78C8, then the offset value would be specified as 6

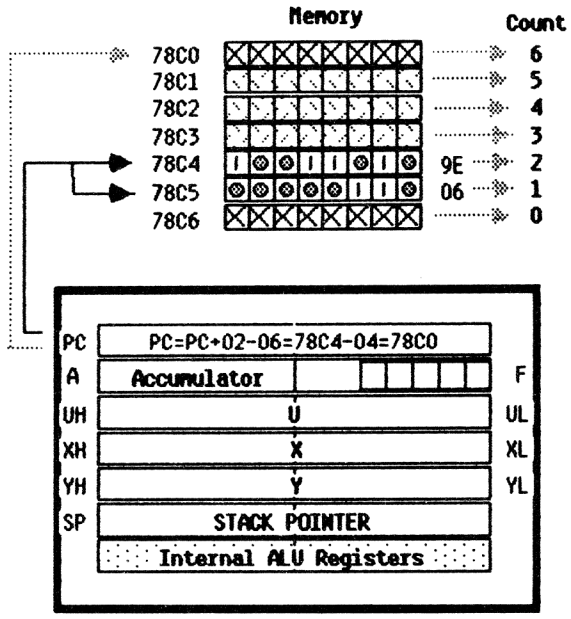


(actually, 06 using hexadecimal notation). Note that the offset value specified is six even though seven bytes of code are being skipped. This is because the first skipped byte is numbered zero. Do you know why this convention is used? It is because the program counter will already have advanced to the next byte of memory by the time it is ready to inplement the offset value! Makes sense, right?

Be careful when it comes time to specify a *reverse branch*. You can still start counting at the byte following the offset value, but you must again count this as the zero byte and you progress backwards (towards lower memory addresses) from that point. Thus, if the reverse branch opcode (9E)

 Copyright 1983 POCKET COMPUTER NEWSLETTER

was stored at location 78C4, the offset value was in 78C5, and the program was to go back to an instruction that started at location 78C0, then the offset would again be specified as 06.



CPU

It is absolutely essential that you understand how to specify these offset values in order to effectively apply these most valuable directives! Do not leave this section until you have carefully studied this text and the accompanying diagrams and are confident you understand the concepts.

Perhaps one way to enhance your understanding is to recall how these directives affect the program counter: the offset value is either added to or subtracted from the program counter, thereby immediately causing it to point to the new location at which to find the next instruction. Now, what value will the program counter have in it at the time it finishes executing a branch directive? Why the address of the byte that follows the current instruction. This has to be the case because the program counter is always telling the CPU where to obtain its operational information.

In the case of a forward branch, advancing the value in the program counter by whatever number of bytes the programmer wants to advance, less one to account for the byte assigned the count of zero, will cause it to be pointing to the desired location. This is easily demonstrated by assuming that rather than skipping anything, the CPU is to simply execute the next instruction stored in memory. In this case the branch offset value would be zero. It would be exactly the same thing as though the program counter had merely advanced itself to the start of the next instruction, as it normally does, when it had finished executing the instruction! Naturally the FB 00 (forward branch with a zero offset value) directive, while it can be specified, has little practical value. The example may, however, help illustrate how the instruction works.

In the case of the reverse branch, where you want to go back a certain number of steps, you have to remember that the program counter is already pointing to the next instruction it would normally execute if the branch did not occur. The offset value will now be *subtracted* from the value in the program counter. If the count of two is subtracted from the program counter at this point, what will happen? Right. The program would instantly be caught in an endless loop. This is because the hapless program counter would be reset to point to the opcode of the reverse branch instruction itself. It would repeatedly perform the same directive over and over again. Remember this well: *a reverse branch with an offset value of two is a sure way to produce CPU lockup!*

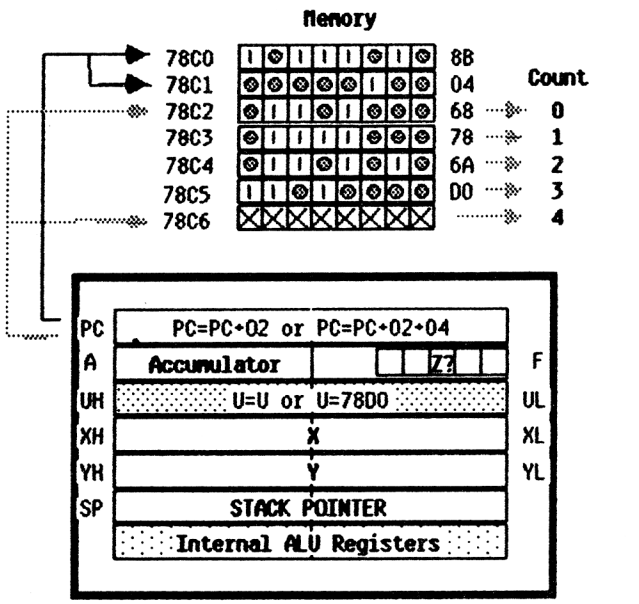
In summary, a plain forward or reverse branch is merely a shortened form of a jump directive. It simply uses *offset addressing* rather than direct addressing. It is, however, the many various forms of the forward and reverse branching instructions that give the LH5801 CPU its decision-making capability.

Flagging a Choice

Remember all those CPU flags we discussed earlier? Now is when you learn how essential they are in controlling the operation of a computer!

Those flags can be "tested" by instructions, such as the branch group being presented here. That is, as part of the operation of a particular instruction, the status of a flag can be examined to determine its current state. Then, depending on the state found, the final operation of the current instruction can be altered. In the case of the branch directives, the alteration to the operation is as follows: If a tested flag is in a desired state (the test condition is *true*), then the branch procedure will take place, *otherwise it will not!* Naturally, if a particular branch instruction is not executed, that is, if the branch (jump) does not take place, then the instruction that follows the branch will be executed *instead of being branched (jumped) over!* Presto! You now have the capability of changing the path of instructions that will be followed *depending on the flag conditions that exist at the time a program is actually being performed.* Great stuff!

Take, for illustration, the *conditional forward branch if zero* directive (represented by the mnemonic FBZ). Again, purely for illustration, let us suppose that the offset value accompanying the opcode was 04, so that the computer would branch ahead *over* the four bytes (to the *fifth* byte) if the



CPU

command was executed. And, to complete the picture, imagine that the instructions LDUH #78 and LDUL #D0 (which are each two-byte directives) followed the branch directive. What would happen as this series of commands was encountered by the CPU?

The FBZ #04 would be encountered. The CPU would examine the status of the zero flag. If it was *set (true)* then a count of four would immediately be added to the program counter. *This would cause the computer to skip over the next four bytes of information in memory and continue executing instructions at the fifth byte.* Thus, the LDUH #78 and LDUL #D0 directives would *not* be performed *if, and only if,* the zero flag was found to be set at the actual time that the branch instruction was executed. On the other hand, if the CPU found that the zero flag was *cleared (false)* at this time, then *the branch would not take place.* Instead, the program counter would merely advance in a normal fashion thereby causing the LDUH #78, LDUL #D0 directives to be performed.

You have now seen how a branch directive may be used to cause something to be done or not done. In this example, the 16-bit U register would be set to point to address 78D0 if, and only if, the zero flag had been set when the branch directive was encountered. Otherwise, whatever previous value had been in the U register would be left there.

The converse instruction is also available as part of the LH5801's repertoire. That is, there is a directive: *forward branch on non-zero* (FBNZ). This instruction is exactly the opposite of the one just described. If the zero flag is cleared (i.e., the register that controlled the flag was non-zero), then the forward branch *is* performed. If the zero flag is set, then the forward branch is *not* taken.

There are six other conditional forward branch instructions that perform similarly, based on the status of the carry (C), half-carry (H) and overflow (V) flags. Thus, there are a total of nine variations of the forward branch directives: one *unconditional* and eight *conditional.*

Mnemonic	Code
FB #nn	8E
FBZ #nn	8B
FBNZ #nn	89
FBC #nn	83
FBNC #nn	81
FBH #nn	87
FBNH #nn	85
FBV #nn	8F
FBNV #nn	8D

Similarly, there are a total of nine variations of the *reverse* branch directives: one *unconditional* and eight *conditional.*

Mnemonic	Code
RB #nn	9E
RBZ #nn	9B
RBNZ #nn	99
RBC #nn	93
RBNC #nn	91
RBH #nn	97
RBNH #nn	95
RBV #nn	9F
RBNV #nn	9D

Note that the only difference between the forward and reverse branch directives is the direction in which the jump occurs. Forward branches move ahead in memory (resulting in a higher address value in the program counter). The reverse branches move backwards in memory (resulting in a lower address value in the program counter).

(Readers who are planning on becoming MLP Masters might also note that the only difference in the opcodes is that the reverse group turns on one more bit. Thus the most significant digit in the opcode becomes a 9 instead of an 8.)

Having such a complete set of conditional branch directives eases the task of developing machine language programs. Note, for instance, that having both branch on zero and branch on non-zero directives, is really a convenience. A programmer could obtain the same logical result in a program if only one of those directives was available, through proper positioning of the code that was to be skipped. Having both possibilities present as instructions, however, makes life easier for the programmer. Say "thank you" to the chip designer, please.

 Copyright 1983 POCKET COMPUTER NEWSLETTER

See for Yourself

Let's examine a little program that demonstrates the use of branch directives. Actually this example is a modification of a simple routine that originally appeared in the instruction manual that is supplied with the Radio Shack PC-2. The machine language routine to be presented will cause whatever is in the liquid-crystal display (LCD) to be displayed in inverse format. A flashing display is then made by coupling the machine language directives to a simple BASIC program.

Here is the series of machine language instructions that will be utilized:

Mnemonic	Description
LDXH #76	Load X-high with value 76.
LDXL #00	Load X-low with 00.
LDA (X)	Load accumulator with the contents of the address pointed to by register X.
EORA #FF	Exclusive OR (invert) the bits in the accumulator. (A commonly used programming "trick".)
STAI (X)	Store the inverted value in the address pointed to by X, then automatically increment the value in the 16-bit X register so that it points to the next location in memory!
CPXL #4E	Compare the contents of X-low with the hexadecimal value 4E.
RBNZ #08	If the result of the comparison is non-zero (i.e., X-low is not equal to 4E), *reverse branch* back to the LDA (X) command.
CPXH #77	Compare the contents of X-high with the hexadecimal value 77.
FBZ #04	If the result of the comparison is zero (i.e., X-high is equal to 77), *forward branch* ahead to the RTS command.
LDXH #77	Load X-high with 77.
RB #12	*Reverse branch* to do the LDXL #00 instruction again and go through second half of LCD buffer area.
RTS	Special command that will pass control back to BASIC program.

The machine language routine as just presented is in what is referred to as *assembly language* or mnemonic form. This is also sometimes referred to as a *source code listing*. Note that there is no indication of where the instructions will be stored in memory within the computer nor is the actual opcode used by the CPU even presented. Instead, just the mnemonic representations for the instructions that will be used are listed. This is the method of notation that experienced machine language programmers use as they think about and create routines that will ultimately be executed by the CPU.

Before such a listing can be used by a computer such as the Sharp PC-1500, the assembly listing must be converted to what is known as the machine readable or *object code* form. This process is also known as *assembling* a program. It consist of nothing more than translating the mnemonic representations for the instructions into the actual binary patterns recognized by the CPU and assigning them to specific memory locations (addresses) in which to reside.

The process of assembling a program is readily accomplished using manual methods, particularly when the program is small. All that is required is a lookup table that gives the machine code for each type of instruction that is being invoked. It may also be necessary to ascertain specific addresses, such as when data is being stored or jump instructions are being used. And, in the case of branch directives, it is necessary to calculate offset values. However, it should be noted that all of these processes are essentially mechanical in nature. That is, a computer can readily be instructed as to how to accomplish such translations and perform address calculations. Hence, it should come as no surprise that it is possible to construct an *assembler program* that will process a mnemonic source listing and convert it to a final object code listing. Professional machine language programmers who work at developing machine language routines on a continuous basis almost always use such a program. It relieves them of the tedious task of performing such translations manually.

Alas, students using this tutorial should plan on becoming adept at using manual assembly methods. The routines presented herein will readily yield to such rudimentary methods while providing many useful insights. The development of an assembler program for the LH5801 CPU that would run on a PC-1500 could easily take several person-months to develop, would use a substantial amount of memory, and would likely find a rather limited market.

Here is how the above routine might appear in "assembled" form:

Address	Code	Mnemonic
78C0	48 76	LDXH #76
78C2	4A 00	LDXL #00
78C4	05	LDA (X)
78C5	BD FF	EORA #FF
78C7	41	STAI (X)
78C8	4E 4E	CPXL #4E
78CA	99 08	RBNZ #08

```
78CC    4C 77    CPXH #77
78CE    8B 04    FBZ #04
78D0    48 77    LDXH #77
78D2    9E 12    RB #12
78D4    9A       RTS
```

Does it make sense to you? I hope so! Note that we have (arbitrarily) decided that the code will be stored in memory starting at address 78C0 (using hexadecimal notation). Do you recall that memory locations 78C0 through 78CF are normally used to store the string variable A$ by the PC-1500 BASIC interpreter? Locations 78D0 through 78DF are normally used to store the string variable B$. Do not use these variables while this routine is residing in memory!

If you have Norlin Rober's LMD program (as recommended in the preceding section of this tutorial), you can use the monitor portion to load the above code directly into memory. If this is the case, then you can skip the first three lines of the following BASIC program.

```
1000: POKE &78C0,&
      48,&76,&4A,0
      ,5,&BD,&FF,&
      41
1010: POKE &78C8,&
      43,&4E,&99,8
      ,&4C,&77,&8B
      ,4
1020: POKE &78D0,&
      48,&77,&9E,&
      12,&9A
1030: INPUT "YOUR
      MESSAGE?",X$
1040: CALL &78C0
1050: FOR X=1TO 50
      :NEXT X
1060: GOTO 1040
```

Note that the first three lines of this BASIC program store the desired machine codes into memory. The remaining lines are used to demonstrate the action of the machine language routine. If you do not use lines 1000 - 1020 of this BASIC program (because you use a monitor to load in the machine codes) then *make sure you do not use a RUN command to execute the program!* If you do so, you will simply *wipe out* the machine codes that were stored where variables A$ and B$ are kept by BASIC! This is because BASIC clears out (initializes) those locations whenever it is given a RUN command. Instead, use a GOTO directive -- such as GOTO 1030 -- to execute the BASIC program without initializing the BASIC variables.

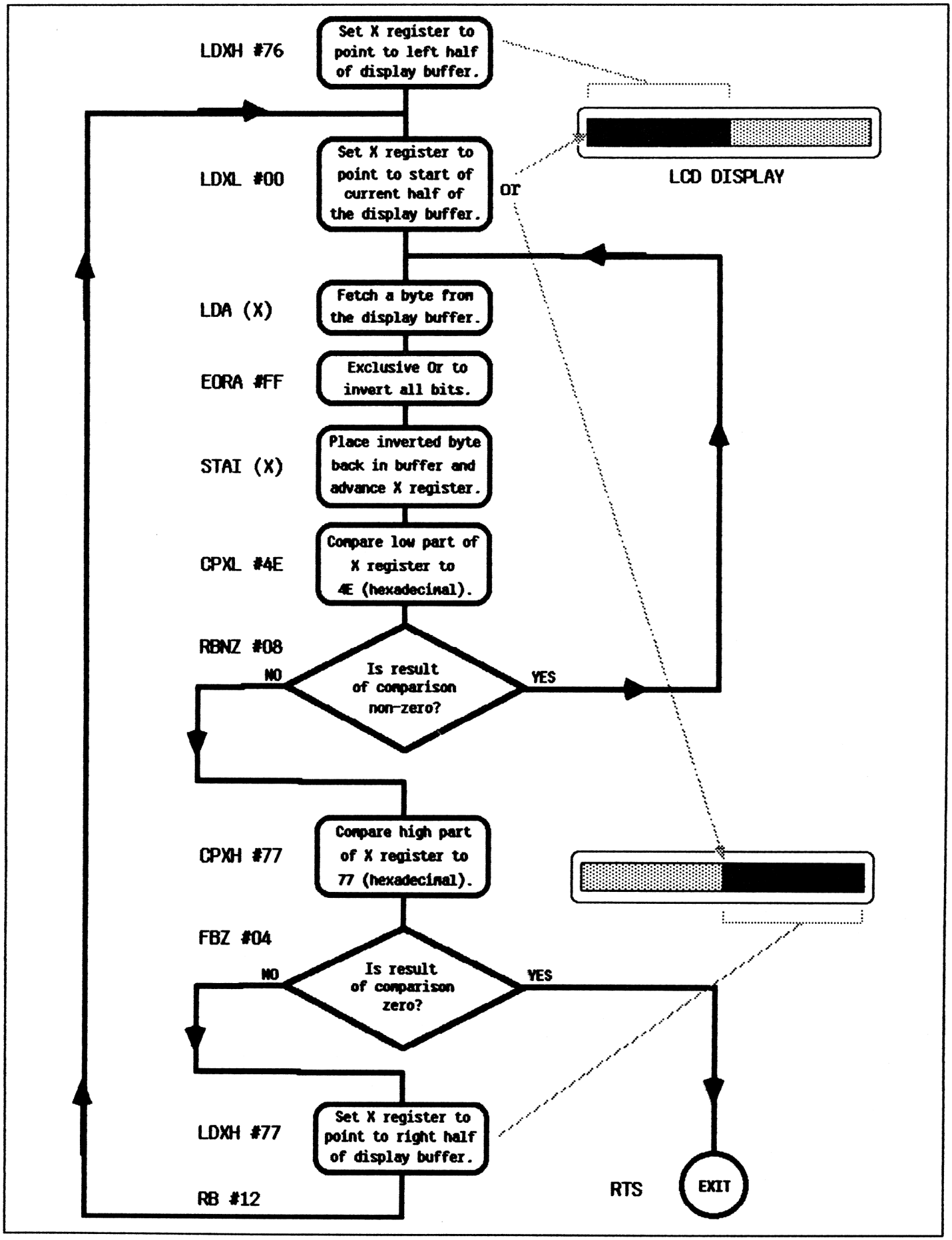
A few words as to what the machine language routine does is undoubtably in order at this point.

You see, the PC-1500 maintains a display "buffer" (a holding zone) in locations 7600 through 764D and 7700 through 774D in memory. This effectively splits the LCD into left and right halves with each block of memory serving one half of the display. (This is a simplification. The addressing of the display is actually more complicated. But, considering it as consisting as right and left halves will do for the sake of this discussion.) The machine language routine being demonstrated does the following:

It sets the address 7600 (the start of the left half of the display buffer) into the 16-bit X register of the CPU. It then fetches a byte of data from the address pointed to by the contents of the X register into the accumulator. That byte of data is then *inverted* using an exclusive OR instruction "against" a byte where all the bits have been set to a logic one state. The inverted data is then stored back into the same memory location in the display buffer. However, the STAI (X) instruction, you may recall, automatically increments the contents of the X register so that it points to the next location in the display buffer (in this case).

Since the first (left) half of the display buffer ends at address 764D and we do not want to alter memory beyond that location, a test is used to determine when that address has been exceeded. This is accomplished by comparing the lower byte of the X register against the value 4E (the value immediately after 4D). This is the value that the lower part of the X register will contain as soon as the STAI (X) instruction has finished executing (when the X register contains the address 764D). Remember that a compare directive simply sets the CPU flags based on the results of the compare. Thus, if the comparison of X-low against the value 4E is non-zero, then the CPU "knows" (in this situation) that the X register has not reached the value 764E. The RBNZ #08 directive thus routes the CPU via the program counter back to execute the LDA (X) directive (at address 78C4 in the assembled listing). A program loop has effectively been formed. This loop or series of instructions will be repeated until X-low attains the value 4E. At that point, with the comparison being equal to zero, the reverse branch will not take place. Instead, the CPXH #77 directive (at address 78CC) will finally be executed.

This compare determines if the entire buffer has been processed (i.e., both halves) by checking to see if the high portion of the 16-bit X register is set to the hexadecimal value 77. If this is indeed the case, then the routine is finished. A forward branch (FBZ #04) is used to direct the program to a special instruction (yet to be introduced) that will conclude the operation. However, if X-high is not yet 77 (in which case it must be 76 as that is what it


LDXH #76
Set X register to point to left half of display buffer.
LDXL #00
Set X register to point to start of current half of the display buffer.
OR
LCD DISPLAY
LDA (X)
Fetch a byte from the display buffer.
EORA #FF
Exclusive Or to invert all bits.
STAI (X)
Place inverted byte back in buffer and advance X register.
CPXL #4E
Compare low part of X register to 4E (hexadecimal).
RBNZ #08
Is result of comparison non-zero?
NO
YES
CPXH #77
Compare high part of X register to 77 (hexadecimal).
FBZ #04
Is result of comparison zero?
NO
YES
LDXH #77
Set X register to point to right half of display buffer.
RB #12
RTS
EXIT

is initially set to by the program), then it must be set to that value (77) so that the right half of the display buffer can be processed. When that is the case, then the RB #12 (reverse branch) instruction is used to send the program back up to the LDXL #00 instruction at address 78C2. This resets the lower part of the X register too, so that the region from 7700 through 774D can now be processed by the same series of instructions that originally did the processing when the X register held values in the range 7600 - 764D!

So what do you think? There is a little machine language routine with barely a baker's dozen directives and yet look at all the work it is doing! And look at all the details involved. Is it worth learning how to do these types of maneuvers?

To summarize, that little routine essentially scans the contents of the display buffer. As it obtains each byte, it inverts all the bits. Those bits determine whether each dot making up a single column in the display is on or off. Thus, the operation results in the display being switched from normal to inverse mode. (The process is somewhat complicated by the fact that the actual display buffer is spread over several separate sections of memory. Such inconveniences frequently occur in the world of machine language programming.)

To make a flashing display (that alternates between normal and inverse modes), the machine language routine is coupled to a simple BASIC program. (The resulting combination will be referred to as a *hybrid* program, in that it contains both BASIC and machine language coding.) This BASIC program (besides poking the machine codes into memory) allows the user to define a message. (That is, set up the display buffer with something nifty for demonstration purposes.) It then uses the machine language routine to invert the display. This inversion will take place in the blink of an eye because the machine language routine is executed in just a few milliseconds. Next, a BASIC loop is used to insert a time delay. The program then jumps back to invert the display again. This causes an alternating or flashing effect. If the time delay (line 1050 in the BASIC program) was not provided, the program would perform so fast that you would barely be able to see the alternating effect. (You can experiment with the rate of flashing by changing the "TO" value in line 1050.)

Say, have you noted that a machine language branch directive is the equivalent of a GOTO statement in BASIC? Good.

The Super Looping Branch

Ah, the prowess of chip engineers! The game of one-upmanship continues even in the world of microcomputers. The LH5801 CPU can execute a very special type of branch instruction that has special application in the area of forming program loops (a section of a program that is repeated a set number of times). This instruction has the following mnemonic and machine code:

Mnemonic	Code
BNZD #nn	88

The mnemonic stands for *branch on non-zero and decrement*. What this instruction does is test the value in the 8-bit UL register. If it is non-zero, then the value in the immediate byte that follows the opcode will be *subtracted* from the program counter. If it is zero, then the program counter is advanced in the normal fashion so that the next instruction stored in memory will be executed. Regardless of the results of the test of the UL register, the contents of that register will also be decremented. Note that the decrement takes place *after* the test has taken place. Note also, that while the test is for a zero condition in the UL register, this test does *not* affect the status of the zero flag (or any other regular CPU flag).

To see the practical application of this special type of branch directive, suppose you wanted to perform a particular sequence of instructions five times. To use the BNZD instruction, you would first load the value 04 into the UL register. (Can you recall the directive that would accomplish this?) Note that the setting up of the UL register must take place outside of the series of instructions that will make up the program loop. (If this advice is not followed, the computer will get caught up in an "endless loop" situation. This can be most embarrasing for the careless programmer.)

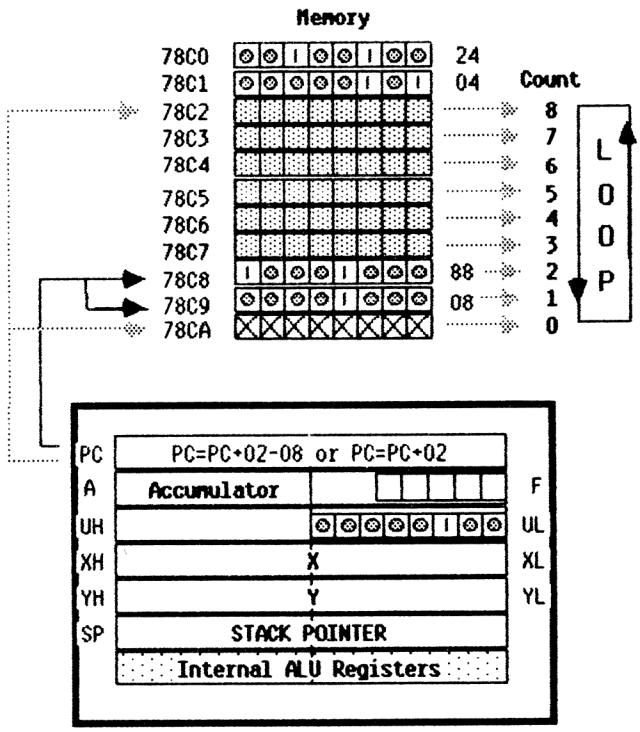
For the sake of illustration, assume the sequence of directives that was to be repeated resided in six bytes. After including the two bytes used by the BNZD directive itself (the opcode and the branch offset value), the proper offset value (immediate data byte following the BNZD opcode) for this example would be 08.

Can you mentally step through the operation of this instruction for the above illustration? Let's do it just to check things out....

The first time the BNZD #08 directive is encountered, the UL register will initially contain the value 04. This is certainly non-zero, so the offset value 08 will be subtracted from the value in the program counter at the conclusion of the operation. This will cause the computer to "loop back" and re-execute the same series of directives starting eight bytes back from where it was at the time it *finished* executing the BNZD command. (Remember, when the CPU has finished executing an instruction, the program counter will have

automatically been advanced the number of bytes consumed by the current directive. Thus it will be pointing to the address immediately following the two bytes used by the BNZD #08 directive itself.) Also, at the conclusion of this first encounter, the value in UL will be decremented so that it is left with a value of 03.

As the BNZD directive is encountered on the second time through the series of directives, the UL register will contain 03 (still non-zero), thus the reverse branch will again be performed and UL will decrease to a value of 02.



CPU

This process repeats until the fifth trip through the "loop" (in this example). This time UL will have a value of zero when the BNZD directive is encountered. This means the branch will *not* be performed. Instead, the program counter will advance normally. The program loop will thus be exited as program execution continues with the opcode contained in the next byte of memory. Note, however, that UL will still be decremented *and will end up with a value of FF!* That is, as the zero value is decremented, the UL register will underflow to the all ones (hexadecimal value FF) condition.

This special looping branch is a powerful directive. The availability of such a command should be considered a luxury by machine language programmers. Earlier types of microprocessors did not have such built-in convenience. You had to roll your own from scratch. (Can you figure out what series of instructions to use to mimic the operation of the BNZD directive?)

The Machine Language GOSUB

Just as you can emulate GOTO statements using jump and branch directives in machine language, there are also instructions that provide subroutine capability at the machine language level. In this tutorial these types of instructions will be referred to as *jump to subroutine* and *call* directives depending on the addressing mode being utilized.

In the BASIC language, a GOSUB statement does the following: the program jumps to a given line number instead of executing the next available statement in the program. However, before skipping off to the designated line, it *saves* its current location (more precisely, the location of the next statement it would encounter had it not performed the GOSUB). It does this so that it can eventually return to resume operations where it had left off before being called away by the GOSUB directive. The return location is saved in a *last-in, first-out (LIFO) stack* that is maintained by the BASIC interpreter.

The program then begins executing a new series of statements starting at the line number indicated by the GOSUB directive. It executes the new series until it encounters a RETURN statement. The RETURN directive causes the last location stored in the LIFO stack to be used as the point to which it is to return. Thus the program "returns" to the point just beyond where the original GOSUB directive occurred and continues its operation.

Machine language programs can also invoke subroutines. If a subroutine is located at a specific memory address, then an all-encompassing *jump to subroutine* instruction may be used to direct the CPU to do the following: save its current location (the address of the next instruction that it would normally execute), then jump to a specific location in memory and begin performing the directives that it finds at that point. How does the CPU do this? You already know that it can jump to a new area in memory by changing the value in the program counter, just as it does for an ordinary jump directive. It can "remember" where it was before going off to perform the subroutine by merely saving the "current" value in the program counter. (The *current* program counter value refers to the value it would have after acting on all the bytes making up the jump to subroutine directive. That is, the address of the byte in memory that follows the bytes used by the jump to subroutine instruction.) Where does it save the value of the program counter? Why in the location pointed to by the contents of the *stack pointer!*

Do you remember what happens when the stack pointer is used by the CPU? Depending on the operation, the stack pointer is incremented or

decremented. When it is used in conjunction with a jump to subroutine directive, it operates as follows: The low portion (least significant eight bits) of the program counter (which is pointing to the third byte of memory after the opcode for the jump to subroutine directive) is stored in memory at the address pointed to by the current value in the stack pointer. The value in the stack pointer is decreased by one. The high portion (least significant eight bits) of the program counter is now stored in memory at the new address pointed to by the stack pointer. The stack pointer value is decremented again so that it is ready to point to the next byte in the current *stack.* In other words the stack is constructed from the "top" of memory downwards. This can be contrasted to the storage of instructions themselves which are executed from the "bottom" of memory upwards. Said another way, the program counter normally advances, while the stack pointer goes backwards as it constructs a stack. (But, it goes forwards when it dismantles a stack!)

Do you understand that a stack in memory created by the operation of the stack pointer can be located in any section of RAM? Did you know that a programmer can create a whole host of stack areas in memory? Do you realize that a program that utilizes the stack pointer must first make sure that the stack pointer is loaded with an appropriate memory address at which to construct a stack? (Yes, there are directives to accomplish this task. They will be presented in due course.) Have you noticed that while the stack pointer seems to operate in reverse, it ends up storing subroutine return addresses in the same order as which they appear when associated with instruction opcodes? That is, the high portion of a two-byte address value ends up in the lower-valued address byte of the stack and the low portion ends up in the stack byte with the higher address.

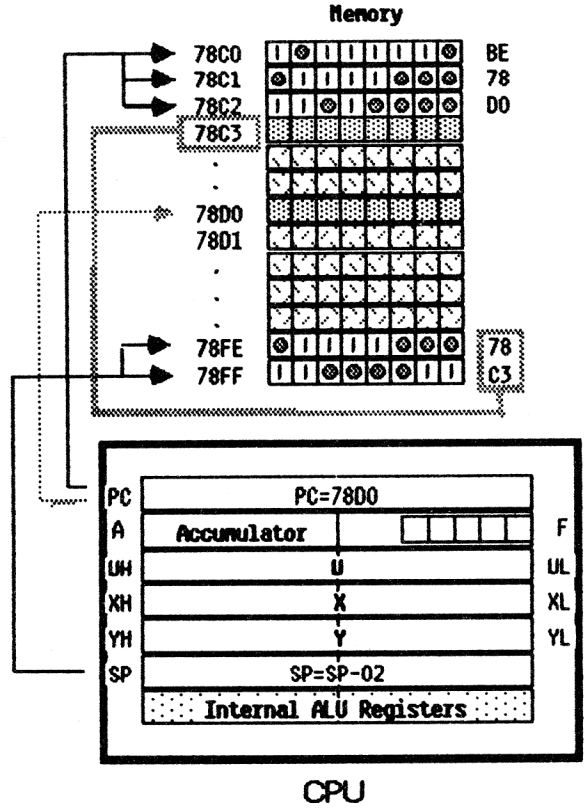
The mnemonic and opcode for the standard *jump to subroutine* directive is:

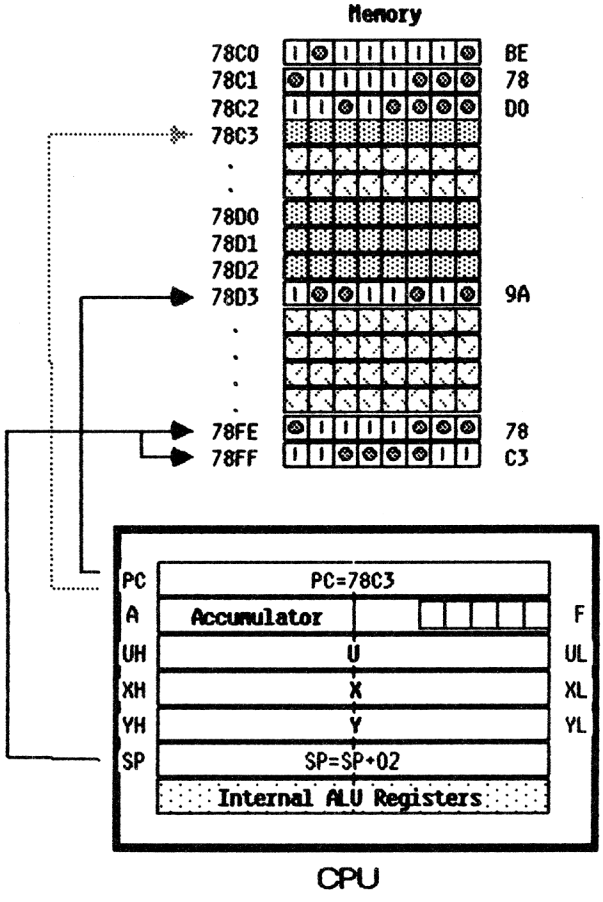
Mnemonic	Code
JSR nnnn	BE

Once the program has jumped to the start of a subroutine, (which must contain a valid instruction opcode), the CPU will continue executing instructions in its normal sequential fashion. It does so until it encounters a special directive that must be used to terminate *all* subroutines. It is known as the *return* instruction. (Wow! Just like in BASIC!) The LH5801 recognizes the following opcode as the standard *return from subroutine* directive:

Mnemonic	Code
RTS	9A

When this command is detected, the program counter is reset to the two-byte address that was saved in the stack when the subroutine was called.





 Copyright 1983 POCKET COMPUTER NEWSLETTER

(Provided that the program has been constructed properly. Read on.)

What actually happens when a RTS directive is found is the following sequence of events: The address in the stack pointer register is increased by one. The contents of the memory location then pointed to by the stack pointer is loaded into the left-most (most significant) half of the program counter. The stack pointer value is increased again by one. The contents of the memory location it then points to is fed into the least significant half of the program counter. The CPU resumes program execution at the address that has just been loaded into the program counter.

Note that if the last thing the stack pointer did previously was to load a two-byte value (address) into a stack in memory (such as occurs when executing a jump to subroutine directive), then a RTS directive will result in that saved address being placed back into the program counter. The stack pointer is also restored to its previous value. The LIFO (last-in, first-out) stack process has been completed.

You might also like to take note of the fact that this orderly process can be subverted by a careless programmer. If for example, the stack pointer is used for some other purpose (and is not properly restored) during the operation of a subroutine, then it will not be able to restore the proper return address to the program counter when an RTS directive is encountered. The execution of a RTS instruction, will however, still cause the program counter to be loaded with whatever two bytes the stack pointer happens to be pointing to. Woe be it to the haphazard programmer who allows such a situation to occur. The result is invariably instant chaos and a totally "bombed" program!

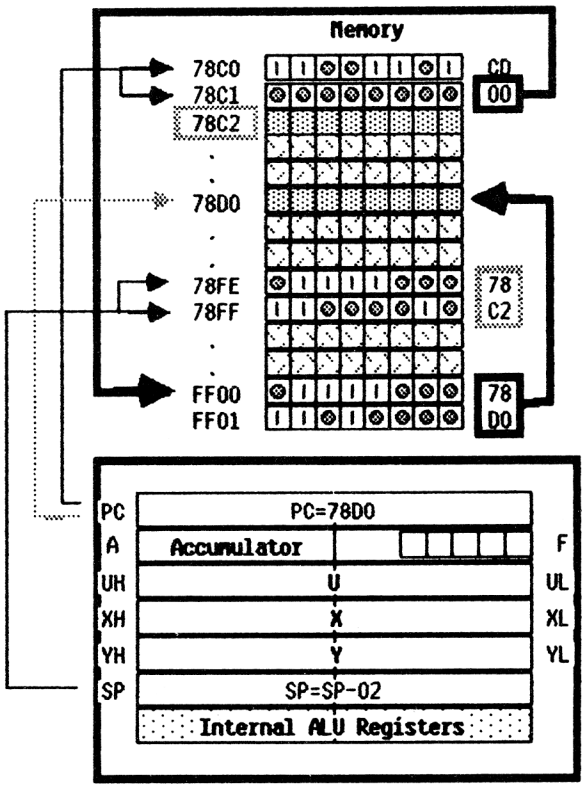
The use of stacks and their control by the stack pointer register is of great importance (besides their use in keeping track of subroutine return addresses). There will be a special section devoted to this particular aspect of machine language programming further on in this series.

Special Subroutine Calling Instructions

The LH5801 CPU can execute several different forms of the fundamental subroutine call directive. Unfortunately, for many PC users, the availability of these instructions will be primarily of academic interest. The reason for this will become clear shortly.

One secondary class of call directives that the LH5801 can perform uses what is referred to as *base page indirect addressing.* Instead of the opcode for the instruction being followed by two bytes containing an absolute address (of where the subroutine starts), it is followed by a single byte. This byte points to the location in a specific block of memory (the *base page*) where a two-byte absolute address value may be found. The base page used by the LH5801 is referred to as page FF. It is the 256-byte block of memory having the hexadecimal address range FF00 through FFFF. Note that it encompasses all the addresses wherein the two most significant digits (FF) of the address do not change, hence the reference to the FF *page.*

In the LH5801 the immediate byte that follows the opcode for this class of call instruction must contain an "even" value, i.e., 00, 02, 04,... FC, FE. This value provides the location *on page FF* of the first byte (high-order portion) of a two-byte address value. It is this address value to which the CPU is to jump and begin executing a subroutine.



CPU

Note that since two bytes are required for each address on page FF and since there are 256 bytes on that page, this type of call directive can only refer to one of 128 different memory addresses stored on page FF at any given time.

Now here is the reason why this type of directive will be of purely educational benefit to many PC users: In the Sharp PC-1500 and Radio Shack PC-2, the address range FF00 - FFFF is occupied by ROM! It is filled with address values that refer to subroutines used by the BASIC package that is similarly installed in ROM. Since these locations cannot be altered by a machine language programmer, they will not be of any use to someone doing machine language programming.... Unless, that programmer can make use of some of

the routines contained within the BASIC ROM itself. (That is not an impossible idea, but it is beyond the scope of this present discussion.)

Never-the-less, this type of call instruction does exist. Indeed, it exists in unconditional and conditional (controlled by flags) form as given in the following compilation:

Mnemonic	Code
CALL #nn	CD
CAZ #nn	CB
CANZ #nn	C9
CAC #nn	C3
CANC #nn	C1
CAH #nn	C7
CANH #nn	C5
CAV #nn	CF

The conditional forms are similar in concept to those described for the conditional branch group. If the flag condition being tested for is met, then the subroutine call is performed, otherwise the CPU ignores the directive and simply performs the next instruction in the current series.

Remember too, that when a call directive is performed, a return address is saved on the stack. The return address is the address of the byte that follows the two bytes used by the instruction (opcode and location on the base page). Study the accompanying diagram to recall, if necessary, how the contents of the program counter are saved at the address pointed to by the stack pointer, just as is done when a directly addressed jump to subroutine (JSR) instruction is performed.

From a practical viewpoint, these two-byte call directives cannot be used unless: (1) You plan to make use of some of the subroutines stored in the pocket computer's BASIC ROM or (2) You plan to unsolder a LH5801 and build your own computer so that you can put the indirect addresses you want in the FF00 - FFFF block of memory!

However, it is worth discussing why these types of instructions are worth having, at least from a programmer's perspective. What is the primary reason for having subroutine capability built into a CPU? It is to conserve on the amount of memory required in a system by permitting the repeated use of selected blocks of code!

As code is developed for a large program, it usually develops that a number of algorithms or procedures are repeated over and over again, but at different points in the program. Instead of having to always insert the same block of code at each place where a duplicate procedure must take place, the experienced programmer will simply place the key series of instructions at a convenient location in memory and terminate it with a RTS (return from subroutine) directive. The starting address of that key block of instructions (subroutine) is then placed as the address bytes of a JSR command whenever it is desired to perform that operation. The net result is the saving of a lot of memory.

Suppose a particular algorithm occupies 50 bytes of program memory. If it were repeated 20 times within a program, it would consume 1000 bytes of storage. Placing the 50 bytes of code into a subroutine (adding one byte for the terminating RTS command) and jumping to it as a subroutine (requiring just 3 bytes of code each time it was referenced) would reduce the memory storage requirement fromm 1000 to just 111 bytes. It is a powerful, essential, programming concept.

The two-byte call directive just described can reduce the overhead even further. As an exercise, try determining how much memory the above example would consume if all references to the subroutine were through a two-byte call directive. (Don't forget the two bytes consumed by the address that would have to be stored in page FF.) Do you come up with a total of 93 bytes?

In fact, the example just given is quite conservative in practice. In a cursory study of the code stored in the ROM used in the PC-1500, it appears that there are between 1,000 to 2,000 references to subroutines within the roughly 16,000 bytes of coding. If it were not for the extensive use of subroutines, the BASIC ROM coding might well occupy 128K of memory! Just the use of the two-byte call directive over the three-byte JSR probably saves about 1,000 bytes of storage. That represents about a six percent reduction in memory requirements.

With that concept in mind -- that the use of subroutines saves a lot of memory space -- you probably will not be surprised to learn that the LH5801 has still another type of jump to subroutine (call) directive. It is a *one-byte* call!

How does it do this? The LH5801 CPU simply recognizes the following opcodes as special call directives using a rather nifty relationship:

Special One-Byte Call Opcodes

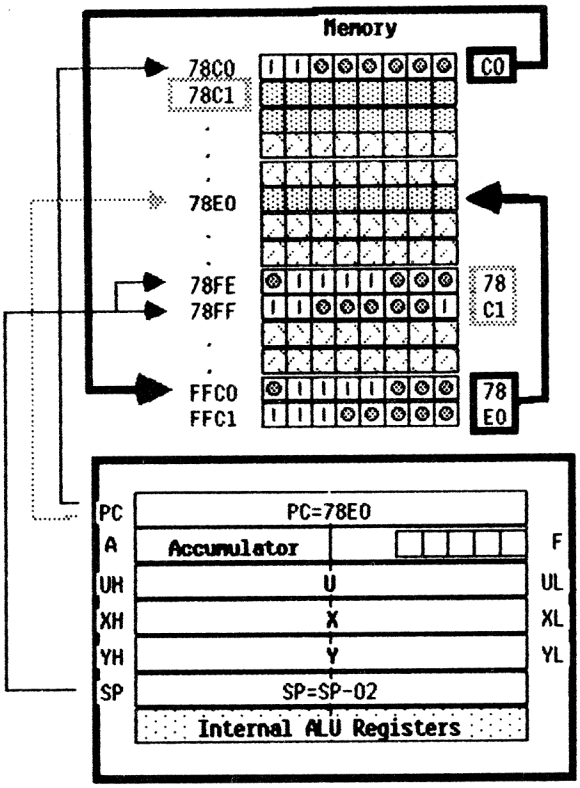
C0	C2	C4	C6	C8	CA	CC	CE
D0	D2	D4	D6	D8	DA	DC	DE
E0	E2	E4	E6	E8	EA	EC	EE
F0	F2	F4	F6	F8	FA	FC	FE

The special opcode relationship is this: the opcode itself serves as the reference to the address on page FF where the absolute two-byte address of the subroutine is stored!

Note that only the upper quadrant of the base page (FF) can be reached by these directives. Of course, just as in the case of the two-byte call, all the locations are "even-valued" as the odd-valued locations contain the second part of each two-byte address that is stored on the base page.

But, these directives all operate in the same

 Copyright 1983 POCKET COMPUTER NEWSLETTER



CPU

manner as the jump to subroutine directive. When executed, the address of the byte that follows the opcode will be stored on the stack as the "return to" address. It is a pretty clever little directive, one not found on typical CPUs. And, in the case of its application in the Sharp PC-1500, its use ends up saving a significant amount of memory.

For instance, working with the example cited earlier (20 calls to a 51-byte block of code), the use of this type of directive would reduce memory storage requirements to just 73 bytes. (Don't forget the two-byte address stored on the base page when considering this case!)

Alas, however, from a practical standpoint the one byte call directive will be of little use unless you can get your hands on a loose LH5801 or you plan to tap into the ROM coding already present in the PC. It is worth knowing about for just such an eventuality, however!

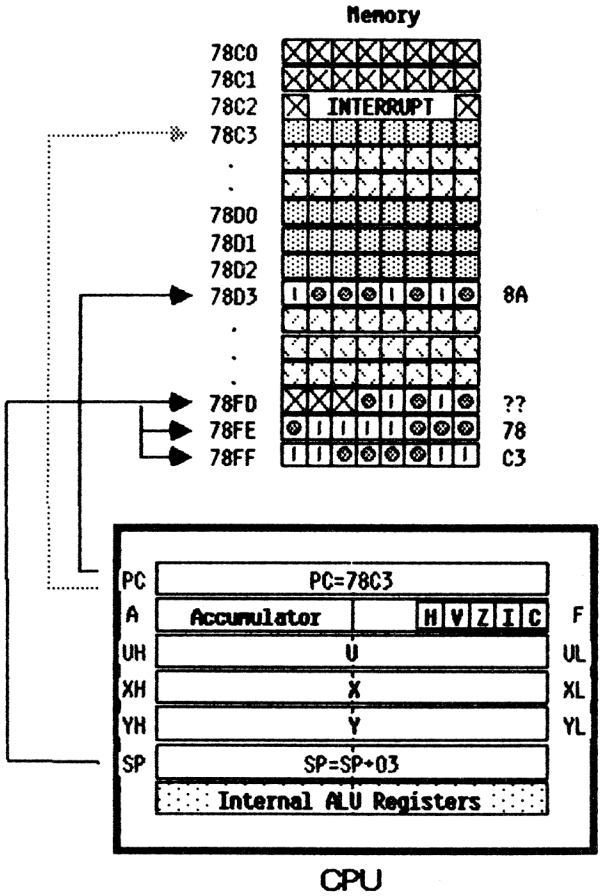
A Special Return Instruction

When certain types of external devices need to communicate with a computer they generate what is known as an *interrupt signal*. The LH5801 CPU responds to such a signal by automatically performing a special kind of jump to subroutine operation. That is, it stops whatever it is currently doing (at the conclusion of whatever instruction it is executing) and pushes the address of the next instruction it would normally execute onto the stack. It also places one more item of information in the stack: the contents of the CPU flag (F) register! It does this so that the present state of all the CPU (programmable) flags can be preserved. Finally, it jumps off to one of several addresses that are stored in the upper locations of page FF. Just which one is dependent upon the type of interrupt that is being performed. The details of this selection will be reserved for later discussion.

It then proceeds to execute whatever instructions it finds at the address it has been directed to as though it were a subroutine. That is, it keeps going until it finds a return instruction. However, since it stored three bytes of information on the stack, it needs to perform a special kind of return procedure in order to return to the place in memory from whence it was originally operating when the interrupt took place. There is a special return directive that performs this extraordinary procedure:

Mnemonic	Code
RTI	8A

The mnemonic stands for "return from an interrupt subroutine." When it is executed, the CPU will first increment the value in the stack pointer, then fetch whatever is in the memory location pointed to by the stack pointer and place it into the CPU flag register. *This action affects all of the CPU flags by restoring them to whatever state*



CPU

they had when the interrupt caused them to be stored! The CPU then increments the value in the stack pointer again, fetches the high part of the return address from the stack, increments the stack pointer once more and fetches the low part of the return address. The return address obtained is placed into the program counter, thereby causing the CPU to resume activity with the next instruction in memory after the one being performed when the interrupt originally occurred.

In summary, the RTI instruction is just like a RTS except that it takes one more byte off the stack. This extra byte is used to set the states of the various CPU flags. Thus, while a regular return directive never alters CPU flags, all flags will be affected by a return from interrupt command.

Since the use of interrupts is strictly limited to special situations dealing with external hardware, many programmers will never have to be concerned with its practical application. Just make sure you don't use it accidently, however. Else you could be in for all kinds of surprises caused by a misaligned stack pointer and a revised set of flag conditions!

Decimal Addition

One of the first instructions I introduced in the first part of this series was the regular old binary addition directive. The LH5801 CPU is capable of performing another type of addition directive known as decimal adjusted addition. It is somewhat more complicated than an ordinary binary add. Hence an explanation of its operation was not included in the previous discussion. After all, I didn't want to scare you off when you were just getting started. If you are still with me at this point, I figure you have enough gumption to deal with just about anything dished out!

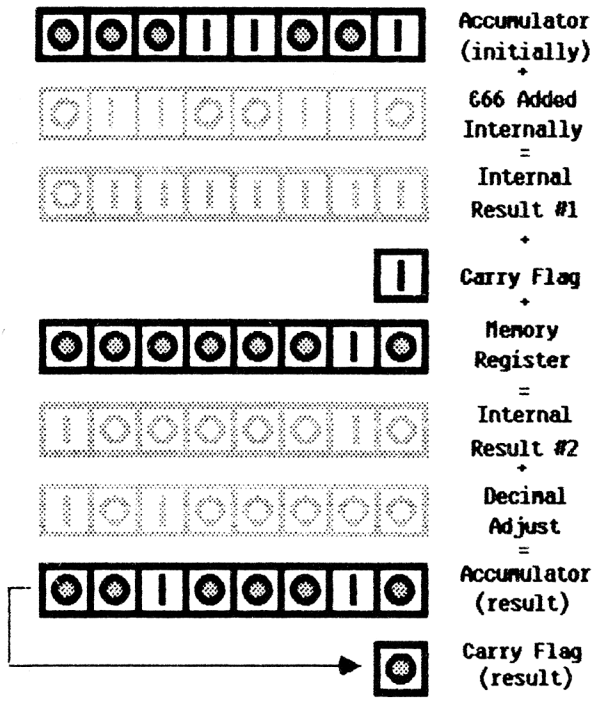
Basically the way a decimally adjusted operation works is to divide a byte into two parts consisting of four bits each. (Four bits are sometimes referred to as a nibble in data processing lingo.) Each 4-bit half-byte is then treated as a decimal value. That is, it is "adjusted" (during the operation of a "decimal" instruction) so that the four bits represent values in the range 0 through 9. Any carry from this procedure is either reflected in the H (half-carry) flag or C (carry) flag depending on which nibble is involved.

The essence of all this is that you can effectively work with numbers in decimal format (actually, binary coded decimal [BCD] format) instead of always having to work in strict binary notation. It turns out that there are drawbacks to working in this format so that it is not as easy to use as it sometimes appears at first glance. However, some programmers like to work with BCD procedures. The LH5801 has the fundamental decimal-oriented instructions that lets those so-inclined work in this mode. Here, for starters, are the three decimal add directives that allow the contents of the memory location pointed to by the U, X or Y registers to be decimal added to the accumulator:

Mnemonic	Code
DADA (U)	AC
DADA (X)	8C
DADA (Y)	9C

And here is the formal explanation of how they operate: (1) The hexadecimal value 66 is added (internally) to the initial contents in the accumulator (which is assumed to be in BCD form), (2) The initial content of the carry flag (C) is added to the least significant decimal digit position, (3) The value in the location pointed to by the data pointer register (which is assumed to be in BCD form) is added to the accumulator, (4) The intermediate result obtained at this point is adjusted to yield final BCD digits by adding in a value that is a function of the half-carry (H) flag and carry (C) flags, (5) Any BCD overflow from the final result in the most significant BCD digit position will be reflected in the carry (C) flag, (6) The status of the Z and V flags may also be affected by the results of the operation, (7) The original contents of the memory location (addend) are not altered by the procedure.

The decimal adjusting operation performed in step (4) of the above description consist of adding particular values (internally) to the accumulator depending on the states of the H and C flags. *It is essential to realize that these flags are operating in such a manner so as to indicate overflows of*



 Copyright 1983 POCKET COMPUTER NEWSLETTER

BCD values from each nibble during this process. The exact value that will be added to "adjust" the results may be determined from this table:

Carry	Half-Carry	Decimal-Adjust Value
0	0	9A
0	1	A0
1	0	FA
1	1	00

The accompanying diagram representing the operation of a decimal addition instruction should help clarify what often seems to be a somewhat obscure procedure. Whether the actual internal process is understood by the programmer is not really critical. The important concept is to realize that if you start out with values in BCD format, you will end up with values in BCD format, provided you utilize these special BCD directives.

Decimal Subtraction

A similar set of directives exist to facilitate subtraction in BCD format:

Mnemonic	Code
DSBA (U)	2C
DSBA (X)	0C
DSBA (Y)	1C

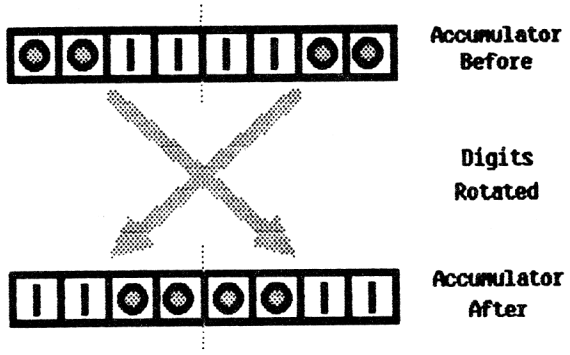
The rules applying to the operation of these directives are as follows: (1) Decimal adjusted arithmetic is performed, (2) The status of the carry flag serves as a borrow indicator at the start of the operation, (3) The value in the accumulator is assumed to be in BCD form at the beginning of the operation, (4) The content of the memory location pointed to by the data pointer register (assumed to be in BCD form) is deducted from the accumulator, (5) The result is decimal adjusted using the status of the half-carry (H) flag and carry (C) flags to determine the adjusting quantity, (6) Any BCD underflow from the most significant BCD digit will be reflected by the status of the carry (C) flag at the conclusion of the operation, (7) The status of the Z and V flags may also be affected by the results of the operation, (8) The original contents of the memory (subtrahend) location are not altered by the procedure.

The adjusting value used in step (4) above, which is dependent upon the values of the carry and half-carry flags at the time of the operation, is the same as that given in the table presented for the decimal addition directives.

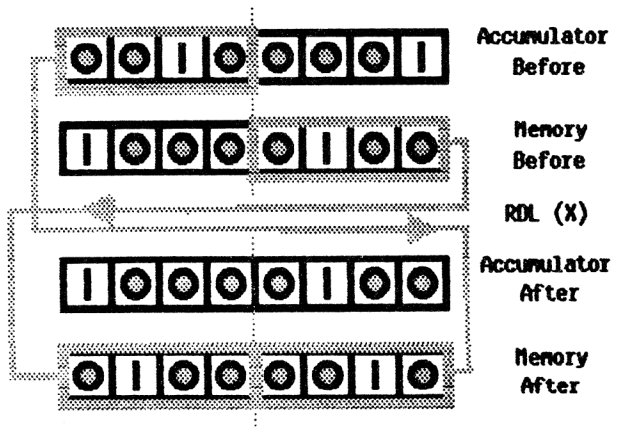
BCD Rotate Instructions

To facilitate working in BCD notation, the LH5801 can also perform several kinds of BCD (or nibble) rotate and swap operations. This permits quick and easy manipulation of BCD digits within the accumulator or between the accumulator and a location in memory.

The simplest of these directives is the RDA (rotate digits in the accumulator) command. This effectively swaps the right and left BCD digits (nibbles) within the A register.



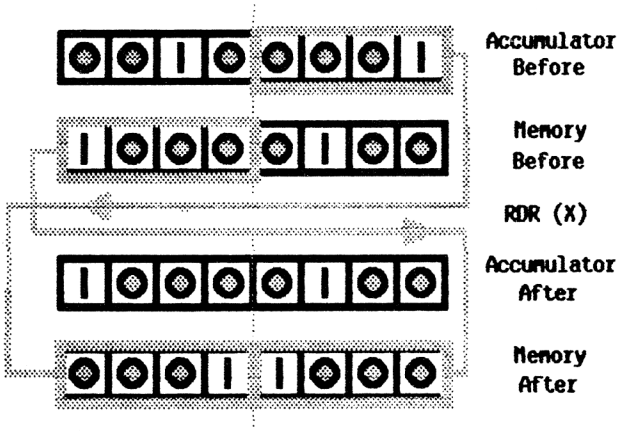
There is also a command that rotates the least significant nibble (BCD digit) of the accumulator into the most significant nibble of the memory location pointed to by the contents of the X register. As this is done, the original most significant nibble of the memory location is shifted to the right, to occupy the least significant nibble position. This BCD digit is also copied into the most significant nibble of the accumulator. Finally, the original least significant nibble of the memory location is transferred to the right-most nibble of the accumulator. The result of all this is called a *digit rotate right* and has the mnemonic RDR (X). The accompanying diagram will help clarify what can be a somewhat confusing operation.



In summary, what effectively happens is that the accumulator ends up containing the original contents of the memory location, while the memory location ends up having the former least significant nibble of the accumulator in its most significant bits and the former most significant nibble of itself shifted over to its least significant bits. *You should note that the original most significant BCD in the accumulator will be lost by the execution of this directive!*

A similar directive with the mnemonic RDL (X)

is used to rotate digits between the memory location pointed to by the address in the X register and the accumulator, in the opposite direction. The most significant BCD digit of the accumulator is passed to the least significant nibble of the memory location. The original most significant BCD digit of the memory location is transferred to the same position in the accumulator. The original least significant nibble of the memory location is shifted left to the most significant nibble position. It is also passed to the least significant BCD digit position in the accumulator. This is the so-called *digit rotate right* command. Again, the diagram can help illustrate the operation.



In summary, the accumulator ends up having the original contents of the memory location. The memory location ends up with its lowest nibble shifted over to the high nibble position. The original high nibble of the accumulator is transferred to the low nibble of the memory register. *The original least significant BCD digit in the accumulator is lost as a consequence of performing this instruction.*

The mnemonics and machine codes for these digit rotate directives are summarized here:

Mnemonic	Code
RDA	F1
RDL (X)	D7
RDR (X)	D3

There is one more important point to remember about the digit rotate instructions: none of the CPU flags are affected by their execution.

A Good For Nothing Instruction

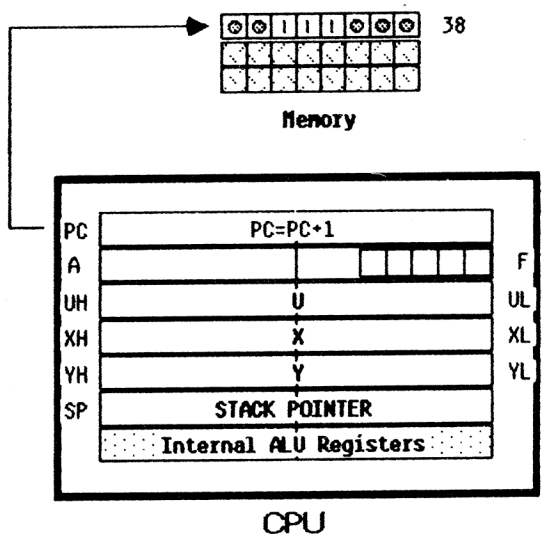
Would you believe there is a instruction that does absolutely nothing? There is. Whenever the CPU sees this directive, it just shrugs it chips, advances the program counter and goes on to the next location in memory.

But, is this "do nothing" instruction really completely useless? No. Most machine language programmers love having it available. It is often used as a "filler" at locations in memory where one suspects programming changes may have to be made. And, it is often used to "patch over" sections of code that are no longer desired in a program.

The instruction is referred to as a *no-operation* (fondly shortened to "no-op") by those in the trade. Its mnemonic in this text is aptly:

Mnemonic	Code
NOP	38

Since the instruction does nothing but allow the program counter to advance to the next memory address, you know that it does not affect the status of the CPU flags.



You will learn just how handy this directive can be the first time you discover you need to replace a three-byte directive with a two-byte one in the middle of a block of machine code!

Switching ROMs or Whatever

The Sharp PC-1500 and Radio Shack PC-2 units can, as you well know, be connected to a variety of peripheral devices, such as a printer or RS-232C communications interface. These devices typically contain ROM(s) (in the address range 8000 - BFFF). Some of them contain several ROMs in this same address range.

You know that the LH5801 CPU cannot possibly execute two different sets of instructions stored in two different ROMs at the same time. In fact, if two ROMs residing in the same memory addresses were activated simultaneously, the result would be electronic chaos. No, the fact of the matter is that the computer must only allow one ROM (within a given address range) to be active at a time. That is, it must tell one ROM when to be active and in so doing tell any competing ROM to cut itself out of the circuit.

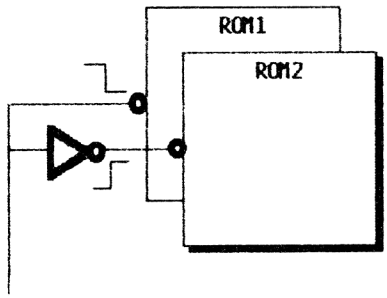
This is a rather straightforward procedure from

 Copyright 1983 POCKET COMPUTER NEWSLETTER

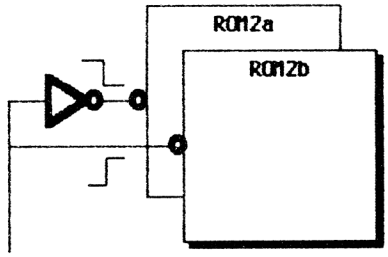
an electronic circuit standpoint. All it takes is a signal that can go from a high state to a low state or the reverse. The output from a settable flip-flop circuit is just what is needed for this chore. Guess what? The LH5801 has several flip-flops (besides the flags that you already know about) that can be controlled by special directives.

Since the PC-1500 uses these flip-flops to control which ROM is activated in an external device, they have been assigned mnemonics that relate to this common use. However it will be pointed out that when the PC is not connected to a device that uses these signals, they could be used for whatever purposes a programmer desired!

One signal line comes out to pin 15 (on the 60 pin connector) of the PC. We will refer to this signal line as the ROM1/ROM2 line. When the logic state of this line is at the low level, it is used to activate ROM1 in an external device. When it is in the high condition, it activates ROM2. You can set this line to the state that activates ROM1 (low level) by executing the ROM1 instruction. Or, you can set it to activate ROM2 (high level) by using the ROM2 directive. Neat, eh?



When 2 ROMs occupy same address range in memory, the ROM1 and ROM2 commands can activate one at a time.



The ROM2a and ROM2b commands can similarly be used to select between another two sets of ROMs.

Similarly another signal line we will refer to as ROM2a/ROM2b comes out to pin 16 of the PC connector. When the signal level on this pin is high it can be used to activate ROM2a. When it is low it switches the external device over to ROM2b. You guessed it: executing the instruction ROM2a puts this line in the high state, while using ROM2b will place it in the low condition. (The RS-232 interface uses this signal to switch between two ROMs that share the address range 8000 - 9FFF.)

Mnemonic	Code
ROM1	B8
ROM2	A8
ROM2a	E1
ROM2b	E3

You will want to recall these instructions the next time you start thinking about how you might control a homebrew external device of your own design. They would seem to provide interesting possibilities in many types of applications.

Lookup Tables

Believe it or not, you have now learned the great majority of all the instructions that can be performed by the LH5801 CPU. There are some more -- those that make up what is referred to as the extended instruction set -- but many of these are simply extensions of those we have already covered. In any event, it is appropriate to spend a little time at this point, getting into the art and science of machine language programming at the practical, hands-on, level.

The first order of business is to understand that it is not necessary to memorize every mnemonic for every instruction that the machine can perform. While some programmers eventually do this, especially after they have worked with a particular CPU for a long time, I do not recommend that anyone put a lot of effort into acquiring such a rote skill. The important thing is to know what classes of instructions -- loads, stores, adds, subtracts, logical operations, etc. -- are available on the machine and what types of addressing modes may be used. This is the conceptual level at which the successful programmer must learn to think.

All of the other aspects of machine language programming may be accomplished by rote table lookup! All you need to have on hand is the appropriate lookup tables. Two such tables are provided on adjoining pages.

The first table is organized according to machine code values. It is used when you want to disassemble machine code. That is, when you need to know what a particular numeric value means to the CPU. Alongside each value that can serve as a CPU opcode is its mnemonic representation. You should note that some numeric values do not invoke a *defined* response from the CPU. *These undefined values should never be used as opcodes within a program.* In other words, the fact that a numeric value is not defined as representing an instruction does not mean that it can safely be used as a NOP (no-operation) directive. It should always be assumed that the use of such a value could produce *unpredictable* results. Hence, they should *never* serve as opcodes within a program.

The second table is organized according to mnemonic representation. It is used when you want to assemble a program. That is, when you want to convert a list of mnemonics (with which the

Lookup Table In Machine Code Order

Code	Mnemonic
00	SUBA XL
01	SUBA (X)
02	ADDA XL
03	ADDA (X)
04	LDA XL
05	LDA (X)
06	CPA XL
07	CPA (X)
08	STA XH
09	ANDA (X)
0A	STA XL
0B	ORA (X)
0C	DSBA (X)
0D	EORA (X)
0E	STA (X)
0F	BITA (X)
10	SUBA YL
11	SUBA (Y)
12	ADDA YL
13	ADDA (Y)
14	LDA YL
15	LDA (Y)
16	CPA YL
17	CPA (Y)
18	STA YH
19	ANDA (Y)
1A	STA YL
1B	ORA (Y)
1C	DSBA (Y)
1D	EORA (Y)
1E	STA (Y)
1F	BITA (Y)
20	SUBA UL
21	SUBA (U)
22	ADDA UL
23	ADDA (U)
24	LDA UL
25	LDA (U)
26	CPA UL
27	CPA (U)
28	STA UH
29	ANDA (U)
2A	STA UL
2B	ORA (U)
2C	DSBA (U)
2D	EORA (U)
2E	STA (U)
2F	BITA (U)
30	
31	
32	
33	
34	
35	
36	
37	
38	NOP
39	
3A	
3B	
3C	

Code	Mnemonic
3D	
3E	
3F	
40	INXL
41	STAI (X)
42	DEXL
43	STAD (X)
44	INX
45	LDAI (X)
46	DEX
47	LDAD (X)
48	LDXH #nn
49	AND (X) #nn
4A	LDXL #nn
4B	OR (X) #nn
4C	CPXH #nn
4D	BIT (X) #nn
4E	CPXL #nn
4F	ADNC (X) #nn
50	INYL
51	STAI (Y)
52	DEYL
53	STAD (Y)
54	INY
55	LDAI (Y)
56	DEY
57	LDAD (Y)
58	LDYH #nn
59	AND (Y) #nn
5A	LDYL #nn
5B	OR (Y) #nn
5C	CPYH #nn
5D	BIT (Y) #nn
5E	CPYL #nn
5F	ADNC (Y) #nn
60	INUL
61	STAI (U)
62	DEUL
63	STAD (U)
64	INU
65	LDAI (U)
66	DEU
67	LDAD (U)
68	LDUH #nn
69	AND (U) #nn
6A	LDUL #nn
6B	OR (U) #nn
6C	CPUH #nn
6D	BIT (U) #nn
6E	CPUL #nn
6F	ADNC (U) #nn
70	
71	
72	
73	
74	
75	
76	
77	
78	
79	

Code	Mnemonic
7A	
7B	
7C	
7D	
7E	
7F	
80	SUBA XH
81	FBNC #nn
82	ADDA XH
83	FBC #nn
84	LDA XH
85	FBNH #nn
86	CPA XH
87	FBH #nn
88	BNZD #nn
89	FBNZ #nn
8A	RTI
8B	FBZ #nn
8C	DADA (X)
8D	FBNV #nn
8E	FB #nn
8F	FBV #nn
90	SUBA YH
91	RBNC #nn
92	ADDA YH
93	RBC #nn
94	LDA YH
95	RBNH #nn
96	CPA YH
97	RBH #nn
98	
99	RBNZ #nn
9A	RTS
9B	RBZ #nn
9C	DADA (Y)
9D	RBNV #nn
9E	RB #nn
9F	RBV #nn
A0	SUBA UH
A1	SUBA nnnn
A2	ADDA UH
A3	ADDA nnnn
A4	LDA UH
A5	LDA nnnn
A6	CPA UH
A7	CPA nnnn
A8	ROM2
A9	ANDA nnnn
AA	LDS# nnnn
AB	ORA nnnn
AC	DADA (U)
AD	EORA nnnn
AE	STA nnnn
AF	BITA nnnn
B0	
B1	SUBA #nn
B2	
B3	ADDA #nn
B4	
B5	LDA #nn
B6	

Code	Mnemonic
B7	CPA #nn
B8	ROM1
B9	ANDA #nn
BA	JMP nnnn
BB	ORA #nn
BC	
BD	EORA #nn
BE	JSR nnnn
BF	BITA #nn
C0	CALL C0
C1	CANC #nn
C2	CALL C2
C3	CAC #nn
C4	CALL C4
C5	CANH #nn
C6	CALL C6
C7	CAH #nn
C8	CALL C8
C9	CANZ #nn
CA	CALL CA
CB	CAZ #nn
CC	CALL CC
CD	CALL #nn
CE	CALL CE
CF	CAV #nn
D0	CALL D0
D1	RRCA
D2	CALL D2
D3	RDR (X)
D4	CALL D4
D5	SRA
D6	CALL D6
D7	RDL (X)
D8	CALL D8
D9	SLA
DA	CALL DA
DB	RLCA
DC	CALL DC
DD	INA
DE	CALL DE
DF	DEA
E0	CALL E0
E1	ROM2a
E2	CALL E2
E3	ROM2b
E4	CALL E4
E5	
E6	CALL E6
E7	
E8	CALL E8
E9	AND nnnn #nn
EA	CALL EA
EB	OR nnnn #nn
EC	CALL EC
ED	BIT nnnn #nn
EE	CALL EE
EF	ADNC nnnn #nn
F0	CALL F0
F1	RDA
F2	CALL F2
F3	

Code	Mnemonic
F4	CALL F4
F5	STI (X)(Y)
F6	CALL F6
F7	CPAI (X)
F8	CALL F8
F9	CLRC
FA	CALL FA
FB	SETC
FC	CALL FC
FD	
FE	CALL FE
FF	

 Copyright 1983 POCKET COMPUTER NEWSLETTER

Lookup Table in Mnemonic Order

Mnemonic	Code
	30
	31
	32
	33
	34
	35
	36
	37
	39
	3A
	3B
	3C
	3D
	3E
	3F
	70
	71
	72
	73
	74
	75
	76
	77
	78
	79
	7A
	7B
	7C
	7D
	7E
	7F
	98
	B0
	B2
	B4
	B6
	BC
	E5
	E7
	F3
	FD
	FF
ADDA #nn	B3
ADDA (U)	23
ADDA (X)	03
ADDA (Y)	13
ADDA nnnn	A3
ADDA UH	A2
ADDA UL	22
ADDA XH	82
ADDA XL	02
ADDA YH	92
ADDA YL	12
ADNC (U) #nn	6F
ADNC (X) #nn	4F
ADNC (Y) #nn	5F
ADNC nnnn #nn	EF
AND (U) #nn	69
AND (X) #nn	49
AND (Y) #nn	59
AND nnnn #nn	E9

Mnemonic	Code
ANDA #nn	B9
ANDA (U)	29
ANDA (X)	09
ANDA (Y)	19
ANDA nnnn	A9
BIT (U) #nn	6D
BIT (X) #nn	4D
BIT (Y) #nn	5D
BIT nnnn #nn	ED
BITA #nn	BF
BITA (U)	2F
BITA (X)	0F
BITA (Y)	1F
BITA nnnn	AF
BNZD #nn	88
CAC #nn	C3
CAH #nn	C7
CALL #nn	CD
CALL C0	C0
CALL C2	C2
CALL C4	C4
CALL C6	C6
CALL C8	C8
CALL CA	CA
CALL CC	CC
CALL CE	CE
CALL D0	D0
CALL D2	D2
CALL D4	D4
CALL D6	D6
CALL D8	D8
CALL DA	DA
CALL DC	DC
CALL DE	DE
CALL E0	E0
CALL E2	E2
CALL E4	E4
CALL E6	E6
CALL E8	E8
CALL EA	EA
CALL EC	EC
CALL EE	EE
CALL F0	F0
CALL F2	F2
CALL F4	F4
CALL F6	F6
CALL F8	F8
CALL FA	FA
CALL FC	FC
CALL FE	FE
CANC #nn	C1
CANH #nn	C5
CANZ #nn	C9
CAV #nn	CF
CAZ #nn	CB
CLRC	F9
CPA #nn	B7
CPA (U)	27
CPA (X)	07
CPA (Y)	17
CPA nnnn	A7

Mnemonic	Code
CPA UH	A6
CPA UL	26
CPA XH	86
CPA XL	06
CPA YH	96
CPA YL	16
CPAI (X)	F7
CPUH #nn	6C
CPUL #nn	6E
CPXH #nn	4C
CPXL #nn	4E
CPYH #nn	5C
CPYL #nn	5E
DADA (U)	AC
DADA (X)	8C
DADA (Y)	9C
DEA	DF
DEU	66
DEUL	62
DEX	46
DEXL	42
DEY	56
DEYL	52
DSBA (U)	2C
DSBA (X)	0C
DSBA (Y)	1C
EORA #nn	BD
EORA (U)	2D
EORA (X)	0D
EORA (Y)	1D
EORA nnnn	AD
FB #nn	8E
FBC #nn	83
FBH #nn	87
FBNC #nn	81
FBNH #nn	85
FBNV #nn	8D
FBNZ #nn	89
FBV #nn	8F
FBZ #nn	8B
INA	DD
INU	64
INUL	60
INX	44
INXL	40
INY	54
INYL	50
JMP nnnn	BA
JSR nnnn	BE
LDA #nn	B5
LDA (U)	25
LDA (X)	05
LDA (Y)	15
LDA nnnn	A5
LDA UH	A4
LDA UL	24
LDA XH	84
LDA XL	04
LDA YH	94
LDA YL	14
LDAD (U)	67

Mnemonic	Code
LDAD (X)	47
LDAD (Y)	57
LDAI (U)	65
LDAI (X)	45
LDAI (Y)	55
LDS# nnnn	AA
LDUH #nn	68
LDUL #nn	6A
LDXH #nn	48
LDXL #nn	4A
LDYH #nn	58
LDYL #nn	5A
NOP	38
OR (U) #nn	6B
OR (X) #nn	4B
OR (Y) #nn	5B
OR nnnn #nn	EB
ORA #nn	BB
ORA (U)	2B
ORA (X)	0B
ORA (Y)	1B
ORA nnnn	AB
RB #nn	9E
RBC #nn	93
RBH #nn	97
RBNC #nn	91
RBNH #nn	95
RBNV #nn	9D
RBNZ #nn	99
RBV #nn	9F
RBZ #nn	9B
RDA	F1
RDL (X)	D7
RDR (X)	D3
RLCA	DB
ROM1	B8
ROM2	A8
ROM2a	E1
ROM2b	E3
RRCA	D1
RTI	8A
RTS	9A
SETC	FB
SLA	D9
SRA	D5
STA (U)	2E
STA (X)	0E
STA (Y)	1E
STA nnnn	AE
STA UH	28
STA UL	2A
STA XH	08
STA XL	0A
STA YH	18
STA YL	1A
STAD (U)	63
STAD (X)	43
STAD (Y)	53
STAI (U)	61
STAI (X)	41
STAI (Y)	51

Mnemonic	Code
STI (X)(Y)	F5
SUBA #nn	B1
SUBA (U)	21
SUBA (X)	01
SUBA (Y)	11
SUBA nnnn	A1
SUBA UH	A0
SUBA UL	20
SUBA XH	80
SUBA XL	00
SUBA YH	90
SUBA YL	10

machine language programmer works and thinks) into the machine codes (numeric values, binary patterns) that the CPU utilizes.

These two tables contain all the instructions that have been presented so far in this text. They will be most valuable to you in the future. Treasure them well!

Initial Program Development

No machine language programmers worth their salt barge into writing a machine language routine. Those that attempt such a brute-force usually end up wasting a lot of time. The number one rule in this kind of programming is: *decide exactly what it is that the computer is to do!* The second rule is: *write it down.*

Now this may seem silly to have to spell out. But the fact of the matter is that many people who learn programming using a high language level such as BASIC fail to realize how rigorous the process becomes at the machine language level. If you make a mistake or change your mind when using BASIC, you can usually just insert a new line here and delete a line or two there to correct the problem. Insertions and deletions don't go over so easily when using machine language.

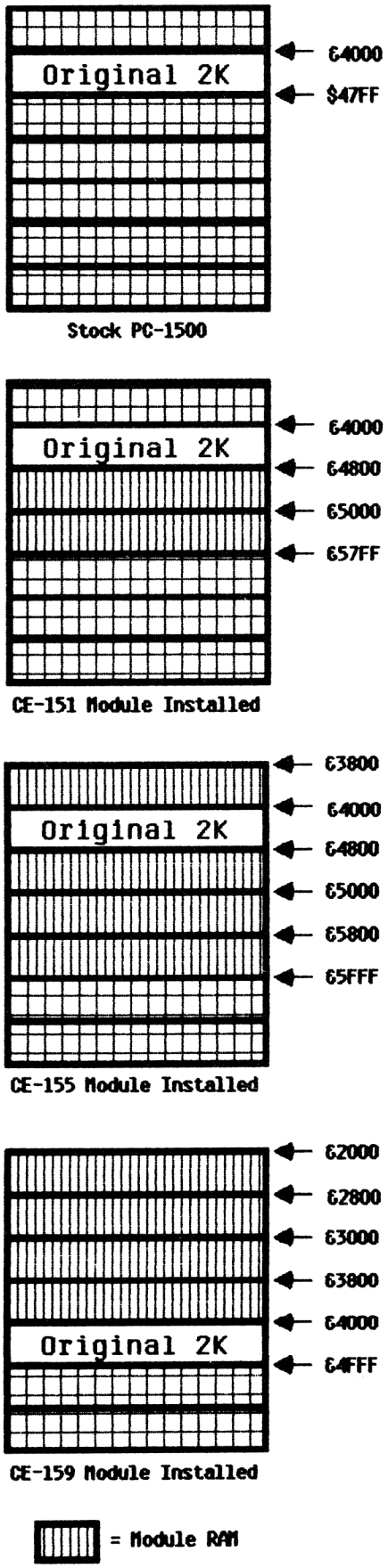
The act of actually writing down, using simple English, the proposed operation of a machine language routine serves several purposes. For one, the process forces a careful review of what you have been planning. This methodical outlining of a process frequently reveals flaws that may have been invisible when the matter was mulled over purely in the slippery recesses of your mind.

Second, this written record can become a guide and checklist as the actual machine language routine is developed. In view of the fact that it may take many hours -- spread over a period of days -- to actually code a substantial routine, having such a written plan to refer to can save a lot of time. It is amazing how much your mind can forget if it is not periodically refreshed. Proper work habits can have a powerful impact on the overall progress one achieves while doing this type of programming.

Of course, if you want to amplify your written descriptions by creating flowcharts, then you should certainly do so. As a general rule, because of the level of complexity involved, the more programming aids you can provide for yourself, the smoother things will go at the machine language level.

Mapping Memory

Another parameter to think about before you start writing a program is where you will store the instructions in memory. You will also need to consider whether there will be any locations used for data storage. It is a good idea to at least rough



 Copyright 1983 POCKET COMPUTER NEWSLETTER

out what is referred to as a *memory map*. This is simply a block diagram that indicates what ranges of memory addresses will be used for specific or general purposes.

The actual locations you use in memory to store a program and/or data will, of course, depend on how much memory you have installed in your PC.

If, for instance, you have a PC-1500 with a CE-155 RAM module installed, then your RAM addresses will run from &3800 - &5FFF. This gives you 10K of RAM. Other modules give other address ranges and amounts of memory as illustrated in the accompanying diagrams.

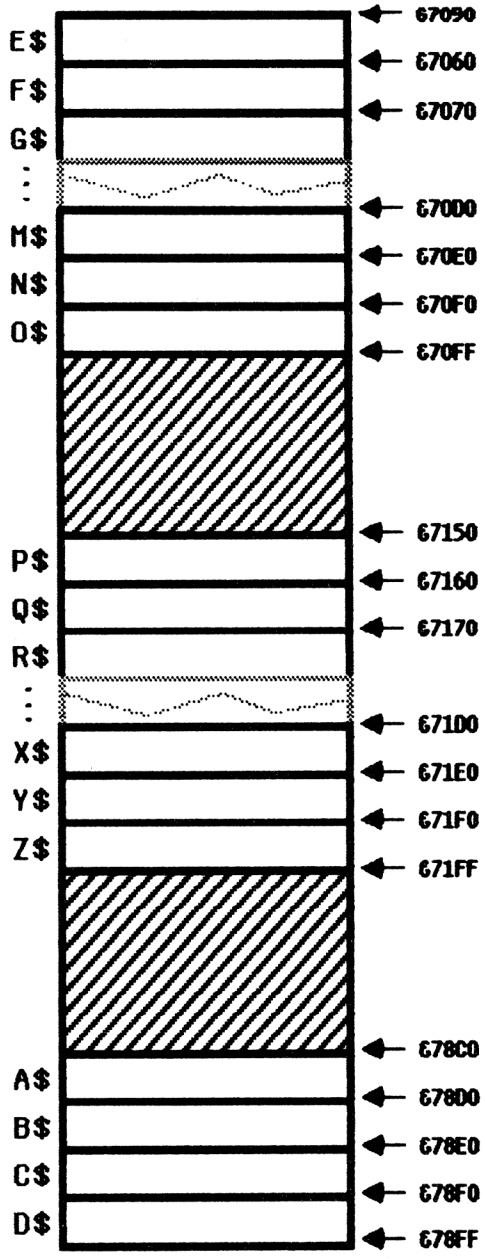
Remember that normally the BASIC interpreter utilizes all of available RAM. However, you can protect (block off) a section of RAM so that BASIC does not use it. This is accomplished by giving the NEW XXXX directive where XXXX represents an address. When invoked, BASIC will not store a user's program below the specified address. Thus, if you have an 8K (CE-155) module installed and you enter NEW &4000, then the area below $4000 (&3800 - &3FFF) will be protected from use by a BASIC program.

If you only plan to deal with small machine language routines, you may want to tuck them away in the special RAM locations normally used to store BASIC variables. Accompanying diagrams identify the address ranges used for such storage. However, if you do use these areas for machine language instructions, it is absolutely vital that you remember the following: *never use any variable whose normal storage location has been usurped for machine language purposes and make sure that you do not use the BASIC statement CLEAR.* If you fail to heed this advice you will find your ML routines suddenly peppered with zero bytes or other unwanted values.

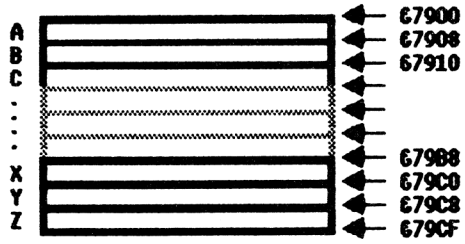
A Machine Language Worksheet

Once you have decided where you plan to store a routine and any associated data and you know exactly what the program is going to do, you can start thinking about what machine language directives to use. As you select each instruction, it is a good idea to write down the mnemonic for the instruction opcode along with any operands or addressing information on a program worksheet. You may also want to assign labels as substitutes for addresses at this point. And, it is a good idea to write down a description of exactly what the instruction will accomplish in relation to the overall program.

A copy of the type of worksheet I like to use for machine language programming is provided for you to make duplicates of, if desired. Development work is done using the right side of the worksheet (in the three columns marked Labels, Mnemonics and Comments). Later, the mnemonics can be "assembled" into machine code and assigned to specific memory locations by filling in columns on the left side of the sheet.



BASIC string variables memory map.



BASIC numeric variables memory map.

How to Assemble a Program

The process of converting the mnemonics used by a programmer into the actual numeric values used by a computer when the program is executed is known as *assembling a program.* A program in mnemonic form is often referred to as the *source code.* The final product of the assembly process is called the *object code.*

The converting of source code (the mnemonics) to object code (the machine code) is a relatively simple process. However, the process can be rather tedious. This is especially true if the program is lengthy or if it is frequently revised. For that reason, most professional machine language programmers like to work with an *assembler program.* This is a program that automatically processes a text file containing mnemonics and translates it to machine code.

Naturally, an assembler program must be especially designed to assemble code for a specific CPU. Since the process involves translating mnemonics into machine codes (table lookup), one of the things necessary for the functioning of such a program is a complete lookup table. Such a table takes a fair amount of memory when stored in the memory of a computer. I estimate that a fully functional assembler program for the PC-1500 might consume some 7 to 8 kilobytes or more of memory. Ten-kilobyte PC-1500 systems are common these days (since the basic 2K PC can be expanded by adding a CE-155 8K RAM expansion module). Even 18-kilobyte systems can now easily be configured (by installing a CE-161 16K RAM module). However, there are still substantial reasons for not attempting to work with a true assembler program in a PC-1500.

One of these reasons is that, when an assembler program is used, it is necessary to pass the source code (mnemonic listing) to the assembler. The ideal way to do this is to store the mnemonic listing as a text file in memory. Of course, to do this you now need storage room for that text file, to say nothing of room for an editor program to create such a file. Finally there is the problem of what to do with the object code created by the assembler program. Again, a nice solution is to be able to store the finished code directly in memory. Alas, this eats up still more of that precious commodity: RAM.

Conventional desktop computer systems often get around the problem of handling source and object code by using what is referred to as a *multiple-pass* system. The source listing is first created using an editor and stored on an external floppy disk. Then the assembler program is loaded into memory. It processes the mnemonic source file from the disk system and stores the object code it produces back on the disk. This final machine code can then be loaded into the appropriate memory addresses and executed as a machine language program. Since a floppy disk can process files rapidly, the entire process of assembling even a relatively large program might take just a few minutes on a desktop system.

Alas, following the same type of procedure using the audio tape storage capabilities of the PC-1500 would consume vast amounts of time. It could take 30 minutes or more just to attempt the assembly of even a small program. This hardly seems practical. Even a beginner can assemble a modest machine language routine using manual table lookup methods in substantially less time. Thus, the manual method will be used in this text!

The process is easy, especially if you write down the mnemonics for the instructions you want to use on an appropriately formatted *worksheet.*

All you have to do is decide the memory address of where you want to start storing the object code. Note this address in the PG (high address byte value) and LC (low address byte value) columns on the worksheet. Now look up the machine code for the mnemonic that is being translated (using the alphabetically-arranged lookup table). Place this code value in the column identified as B1 on the worksheet opposite the corresponding mnemonic. Next, determine whether the instruction uses additional bytes. That is, if it must be followed by addressing information or immediate data. (This information is also available from the lookup table.) If so, insert the necessary information (in numeric form), a byte at a time, in the columns titled B2 through B5.

Once this has been accomplished for an instruction, you can go on to the next line on the worksheet. At this point you can fill in the starting address of the next instruction simply by counting the number of bytes utilized by the preceding directive! Thus, if a three-byte directive was stored beginning at address &7151, then the next instruction would be stored beginning at address &7154.

The only time things can get a little difficult is when the program contains what is known as *forward references.* That is, when it contains jumps , calls or branches to sections of the program that have not yet been assembled. The reason this can cause problems is because you will not initially know the absolute addresses at which such referred-to directives will be stored. Yet, to properly assemble the machine code for such references, you will eventually need to specify this information precisely.

Ah, but the situation is not hopeless. Anytime you encounter such a directive, you simply set aside the appropriate number of bytes needed to

PG	LC	B1	B2	B3	B4	B5	LABELS	MNEMONICS	COMMENTS

complete the reference. In the case of branches, this is always one byte. A jump directive requires two extra bytes to specify an absolute address. A call directive will also usually require two bytes. (You may remember that several special classes of call directives may use just one or no additional bytes. However, since the vectors used by these types of calls are occupied by the BASIC ROM in the PC-1500/PC-2, most programmers will not have occassion to use these special cases.) By set aside, I mean mark the appropriate columns (such as B2 and/or B3) on the worksheet as being reserved. You then count those bytes in order to determine the starting address of the next instruction to be noted on your worksheet. When the actual address of the referred-to instruction is eventually determined, you can go back and fill in the appropriate values in the reserved column locations. A few programming examples in the future will clarify this procedure. It is quite straightforward, though it does require taking care to insure accuracy.

This matter of forward references within a machine language program is the reason programmers like to work with *labels* instead of absolute addresses. The column titled "Labels" on the worksheet is provided for this purpose. Whenever you want to refer to particular points within a program, you can tag them by creating your own reference names. You can then specify references to those points using the label names. Thus, instead of the mnemonic for a jump directive being JMP &7060, you might write JMP THERE (in the worksheet column titled "Mnemonics"). The referred-to point in the program would then be appropriately identified on the worksheet by placing the name of that tag (THERE) in the "Label" column. Again, this methodology will be illustrated in future programming examples.

Loading and Testing a ML Program

Once the machine code for a program has been assembled by hand, it is necessary to load it into the appropriate locations in the memory of the PC.

In the case of a small routine, this can easily be accomplished using the POKE directive that is available in BASIC. The format of this statement is: POKE X,Y where X represent an address and Y stands for the code to be stored therein. There is an alternate format for this statement. The address specification may be followed by a series of data values separated by commas. Thus, a statement such as POKE &7050,&48,&71,&4A,&50 would specify that the hexadecimal values 48, 71, 4A and 50 were to be stored in the four bytes of memory starting at the hexadecimal address 7050.

If you plan on doing a substantial amount of ML programming, then the use of a so-called monitor program can be of great practical value. A monitor program in this context is one that assists the programmer in the process of storing machine codes into memory, examining and altering the contents of memory locations, and so forth. I would recommend that a serious ML enthusiast consider obtaining the Loader/Monitor/Disassembler that is sold by *PCN*. This is an integrated package of routines that greatly facilitates working in machine language on a PC-1500 or PC-2.

Once a ML routine has been stored in memory it may be executed by calling it from a BASIC program. Alternately, a monitor program may be used to check its operation. If a complex routine does not operate as expected, it may be necessary to debug the routine. Debugging ML routines can be very difficult without the assistance of a monitor program. Thus, it is extremely important to plan the operation of a ML routine carefully so as to guard against errors.

An errant ML program will often result in the PC "locking up." The only way to recover from such a state is through the use of the reset button on the back of the unit. As you undoubtably know, using the reset button can cause all of memory to be erased. You then have to start program loading all over. This can be a rather discouraging event. Careful planning and coding of your routines can significantly reduce the chances of this occurring. However, you probably won't consider yourself properly initiated into the realm of MLP until you have had a routine "bomb" and lockup your PC.

The only thing worse than lockup is to discover that you left out an important step in your program somewhere near the beginning. This means all of the instructions beyond that point must be relocated, forward references altered, and so forth. After a few mistakes of this nature, you will gain a profound appreciation for the cultivation of good programming work habits that can help reduce the chances for making such errors.

In the next section of this series I will present a number of practical ML programming routines. In addition to their educational value, these routines can be the start of your own personalized library of frequently used ML procedures and algorithms.

In the meantime, why not make some duplicates of the worksheet form and try your hand at creating some useful ML routines?

Machine Language Programming
the Sharp PC-1500 and Radio Shack PC-2
is published by
POCKET COMPUTER NEWSLETTER
P.O. Box 232, Seymour, CT 06483